Lecture 3: Model APIs, structured outputs, and typed decisions

Patrick Hall

Department of Decision Sciences

November 2, 2026

© 2026–2027 Patrick Hall (jphall@gwu.edu). Some rights reserved. Text and copy are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0).

Agenda

API Requests and Parameters

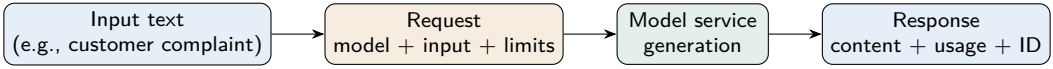
Schemas and data contracts

Typed decisions

Synthesis

References

An API call is a bounded application transaction



A four-stage path separates the application's input and constraints from the model service's response and operational metadata.

- A request chooses a model, supplies input and operating limits, and may attach response-format constraints.
- Preserve the request ID, model, completion state, usage, and application version for diagnosis and accounting; take care when logging sensitive text.
- Keep credentials out of source code: load `OPENAI_API_KEY` from an ignored `.env` file or obtain it interactively.

Source: Responses API fields and lifecycle: OpenAI (2026d).

Inspect the response, not only its visible text

```python
response = client.responses.create(
    model="gpt-5-mini",
    input="Customer says: charged twice...",
    max_output_tokens=250,
)
print(response.output_text)
print(response.id)
print(response.status)
print(response.usage)
```

Python code prints both the visible answer and the operational fields needed to observe one API call.

Illustrative inspection

```
id:  resp_...
status:  completed
input_tokens:  46
output_tokens:  91
visible text:  "I am sorry ..."
```

The generated text may be most interesting to the user, but the ID, status, and usage help an application decide what to retain, retry, charge, or investigate.

Source: Response objects and usage fields: OpenAI (2026d).

Parameters are part of the operating contract

Control	Example value	Purpose	Application question
Model	`gpt-5-mini`	capability, price, context, and supported features	Is this model fit for the task?
Output limit	`800 tokens`	caps visible and internal generation budget	What is a useful maximum response?
Sampling	`temperature=0.2`	varies output diversity when supported	Is variation useful here?
Reasoning	`minimal`	allocates deliberation where supported	Is extra latency worth it?
Timeout / retry	`45 s; 3 tries`	bounds waiting and repeated work	When should the caller stop?
Rate / spend	`60 req/min; daily cap`	protects availability and budget	What occurs under pressure?

Illustrative settings show how an application makes model behavior, resource use, and failure boundaries explicit.

Settings and names differ across models and providers. Treat a request configuration as versioned application code, not as universal model behavior.

Source: Model-specific request capabilities: OpenAI (2026d); rate-limit concepts: OpenAI (2026c).

Compare configurations on the same complaint

```python
import time

def run(label, **settings):
    started = time.perf_counter()
    response = client.responses.create(
        model="gpt-5-mini", input=COMPLAINT,
        max_output_tokens=250, **settings,
    )
    return {
        "label": label, "seconds": time.perf_counter() - started,
        "tokens": response.usage.output_tokens,
        "text": response.output_text,
    }

fast = run("bounded", reasoning={"effort": "minimal"})
careful = run("deliberate", reasoning={"effort": "medium"})
```

A small experiment holds the complaint and output cap constant while changing one supported setting and records time, tokens, and text.

Source: Reasoning and output controls are model-dependent: OpenAI (2026d).

Illustrative results

```
bounded
1.2 s; 74 output tokens
"I am sorry about the duplicate charge ..."

deliberate
2.9 s; 112 output tokens
"The duplicate charge appears to be ..."
```

Both answers may be usable. The deliberate setting used more time and tokens, so retain it only if evaluation shows enough added value for the task.

Streaming improves interaction; it does not change validation

```python
stream = client.responses.create(
    model="gpt-5-mini",
    input="Acknowledge this complaint.",
    stream=True,
)
parts = []
for event in stream:
    if event.type == "response.output_text.delta":
        parts.append(event.delta)
        print(event.delta, end="", flush=True)
final_text = "".join(parts)
```

Illustrative stream

```
[3 s elapsed]
I am sorry about
the duplicate charge.

[10 s elapsed]
I am sorry about the duplicate
charge.  I will help investigate.

final_text
I am sorry about the duplicate
charge.  I will help investigate.
```

A streaming loop makes text fragments available as they arrive while retaining the complete assembled text for the application.

- Stream prose when an early fragment improves the user experience.
- Buffer structured content until complete; then parse and validate it.
- Show a truthful working state; an incomplete stream is not a completed output.

Source: Streaming response events: OpenAI (2026d).

Failure is not one category

Observed outcome	Meaning	Bounded handling
Network timeout or 5xx	service outcome unknown	retry only work safe to repeat; use exponential backoff
429 rate limit	caller exceeded an allocation	honor `Retry-After`; queue or shed load
Refusal	model declined the task	route to a safe alternative or human review
Incomplete response	output stopped before completion	record status; retry or request a shorter result
Schema or Pydantic error	text/data fails the contract	do not execute; repair once or escalate
Business-rule error	valid shape, unacceptable decision	reject and review the policy or evidence

Distinguish transport and rate-limit problems from refusals, incomplete work, structural errors, and business-rule violations.

Retries are explicitly designed and bounded attempts, not an automatic infinite loop. Add a unique ID and log the class of failure without duplicating sensitive content.

Source: Service error classes and rate limits: OpenAI (2026b); OpenAI (2026c).

JSON is syntax; a schema is a contract

Syntactically valid JSON

```
{"category": "billing", "refund": -25}
```

It parses, but a negative refund may violate business meaning.

Schema-constrained object

```
{
  "type": "object",
  "required": ["category", "refund"],
  "properties": {
    "category": {
      "enum": ["billing", "delivery"]
    },
    "refund": {
      "type": "number", "minimum": 0
    }
  }
}
```

The schema specifies fields, types, required values, enumerations, and nested structure.

Two compact JSON examples contrast parsable data with a schema that adds required fields, allowed values, and a numeric range.

Schema-constrained generation can improve format reliability, but it does not establish that a category or amount is substantively correct.

Source: JSON syntax: Bray (2017); schema vocabulary: JSON Schema Organization (2022); constrained generation: OpenAI (2026e).

Extract complaint facts into a constrained object

```
complaint_format = {
  "type": "json_schema",
  "name": "complaint_facts",
  "strict": True,
  "schema": {
    "type": "object",
    "additionalProperties": False,
    "required": ["category", "amount", "evidence"],
    "properties": {
      "category": {
        "type": "string",
        "enum": ["billing", "delivery", "access"]
      },
      "amount": {
        "type": ["number", "null"], "minimum": 0
      },
      "evidence": {
        "type": "string", "minLength": 10
      }
    }
  }
}
```

```
response = client.responses.create(
    model="gpt-5-mini",
    input=COMPLAINT,
    text={"format": complaint_format},
)
payload = json.loads(
    response.output_text
)
```

Example output

```
{"category": "billing",
  "amount": 24.99,
  "evidence": "charged twice"}
```

Schema constraints shape the payload; local Python checks it before use.

The left column defines the data contract; the right column passes it to a Responses API call and reads the returned JSON.

The `evidence` field anchors the result in complaint text. Compare it with the source before acting.

Source: JSON Schema constraints: JSON Schema Organization (2022); provider-specific structured-output request format: OpenAI (2026e).

Pydantic turns a data contract into executable checks

- Pydantic is a Python library for parsing and validating data.
- Typed Python fields become runtime validation and JSON Schema.
- `model_validate` returns a validated object or raises `ValidationError`.
- One schema can support request construction, response validation, and application-side checks.
- Alternatives include direct `jsonschema` checks, Marshmallow, or `dataclasses` with manual validation.

```python
from pydantic import BaseModel, Field, ValidationError

class RefundLookup(BaseModel):
    charge_id: str = Field(min_length=10)
    refund_amount: float = Field(ge=0)

try:
    args = RefundLookup.model_validate(payload)
except ValidationError as error:
    print(error.errors()) # inspect; do not use the payload
```

The code defines a refund-request schema, then turns a payload into validated arguments or reports a validation error.

Source: Pydantic models and generated JSON Schema: Pydantic (2026); schema concepts: JSON Schema Organization (2022).

Structural validation is necessary but not sufficient

```python
class Refund(BaseModel):
    amount: float = Field(gt=0, le=250)
    currency: Literal["USD"]

class Triage(BaseModel):
    action: Literal["request_information",
                    "recommend_refund", "escalate"]
    rationale: str = Field(min_length=10)
    refund: Refund | None = None
```

```python
def business_check(t: Triage):
    if t.action == "recommend_refund" and not t.refund:
        raise ValueError("refund is required")
    if t.action != "recommend_refund" and t.refund:
        raise ValueError("refund is not allowed")
```

Business-rule inputs

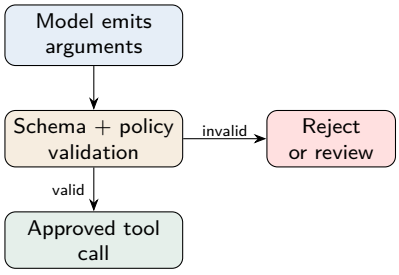
Pass	action:	"recommend_refund"
	refund:	24.99 USD
Fail	action:	"escalate"
	refund:	24.99 USD

A typed triage record and a second business-rule function enforce that a refund exists only for the refund recommendation.

- Field relationships are business rules; state them explicitly.
- The same parsed input always gives the same accept-or-reject result.
- Run this deterministic, application-side guardrail before acting—not as another model judgment.

Source: Data-model validation: Pydantic (2026); structural controls are one component of generative-AI risk management (National Institute of Standards and Technology 2024).

Schemas also make tool boundaries inspectable



Illustrative pseudocode

```
valid = {"charge_id": "ch_482",
         "refund_amount": 24.99}
args = RefundLookup.model_validate(valid)
lookup_recent_charge(args.charge_id) # tool
# accepted: {"status": "pending"}

invalid = {"charge_id": "ch_482",
           "refund_amount": -24.99}
# raises ValidationError: negative refund
RefundLookup.model_validate(invalid)
# no tool call; add to review queue
```

Validation routes model-produced arguments either to a tool call or to rejection and review.

The application validates arguments before invocation; the model neither authorizes nor directly executes the lookup.

Source: Structured outputs and function definitions: OpenAI (2026e); governance context: National Institute of Standards and Technology (2024).

A typed decision makes actions and uncertainty visible

Decision type	Required supporting fields	Example use
`request_information`	missing facts; one question	"What date was the duplicate charge?"
`recommend_refund`	amount; currency; evidence	"Refund $24.99 for the duplicated charge."
`escalate`	reason; evidence; urgency	"Possible fraud; specialist review required."

A decision table gives each allowed triage action its own required evidence, amount, or question fields.

This Jev-style pattern treats a decision as a discriminated set of allowable alternatives, each with the information needed to act or review it. It prevents an ambiguous free-text answer from silently becoming an operational command.

Local option: H2O-Lightning-4B is an open-weight, Apache-2.0 decision model that can run locally; validate it on your own cases.

Source: Practical decision-model guide: TypeSafe AI, 2026, *A practical guide to Jev*; local option: H2O.ai, 2026, *H2O-Lightning-4B*; implementation with Pydantic: Pydantic (2026).

Request a typed triage decision from the API

```python
result = client.decisions.create(
    model="gpt-6-luna",
    input=(
        "Policy: refund duplicate charges; escalate disputed "
        "evidence. Record: two claimed charges; ledger has one."
    ),
    questions=[{
        "type": "choice", "name": "next_step",
        "instructions": "Select the next step.",
        "choices": [
            {"value": "refund", "description": "Recommend refund"},
            {"value": "abstain", "description": "Request more evidence"},
            {"value": "escalate", "description": "Request review"},
        ],
    }],
)
answer = result.answers[0] # choice or refusal
```

Choice answer

```
choice:  "escalate"
probabilities:
  refund:  0.07
  abstain: 0.03
  escalate: 0.90
confidence: 0.88
```

The API selects one supplied option and returns its confidence alongside probabilities for every allowed option.

- Only supplied next-step values can be returned; application code maps the choice to a local workflow.
- A choice answer includes all option probabilities and a confidence value.
- Reported probabilities are model scores, not calibrated guarantees.(Guo et al. 2017)
- Compare performance and cost to traditional statistical/ML classifiers.

Source: Decisions API, choice answers, probabilities, and confidence: OpenAI (2026a); calibration: Guo et al. (2017).

Abstention and escalation are decision outcomes

Complaint

"I was charged twice, and someone changed the shipping address. I need this fixed today."

Typed result

```
{"action":"escalate",
 "reason":"possible account takeover",
 "urgency":"high",
 "evidence":"shipping address changed"}
```

A suspicious complaint maps to an escalation object with an explicit reason, urgency, and supporting text instead of an automated refund.

- `abstain` requests more evidence; `escalate` routes a risky or policy-sensitive case to a specialist.
- A returned choice guides the workflow; it does not authorize or execute an action.
- A deterministic application-side guardrail can override any returned choice and route the case to review.

Source: Decision outcomes and choice answers: OpenAI (2026a).

Evaluate three different claims

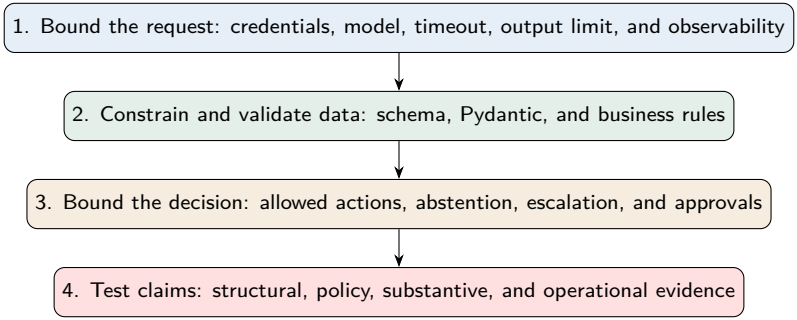
Claim	Check	Example metric
Structural validity	Does output parse and satisfy the contract?	schema-valid rate
Business-rule compliance	Does the object obey policy relationships?	invalid-action rate
Substantive correctness	Is the extraction or decision right for this case?	labeled extraction / decision accuracy
Escalation behavior	Does it defer when it should?	escalation precision and recall

Decision evaluation should address output format, policy compliance, factual decision quality, and escalation behavior.

- A schema-valid decision can still be wrong.
- Keep a labeled, representative holdout set with routine, ambiguous, high-risk, and missing-information cases.
- Lecture 7 contains a fuller treatment of test, evaluation, verification, and validation (TEVV).

Source: Evaluation should support a stated decision and risk context: National Institute of Standards and Technology (2024).

A reliable model API application has layered controls



Four stacked layers show the progression from bounded API requests to validated data, controlled decisions, and multi-level evaluation.

Source: API and structured-output behavior: OpenAI (2026d) and OpenAI (2026e); risk-management context: National Institute of Standards and Technology (2024).

References

Bray, Tim (2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Internet Engineering Task Force. URL: https://www.rfc-editor.org/rfc/rfc8259.

Guo, Chuan, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger (2017). "On Calibration of Modern Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*, pp. 1321–1330. URL: https://proceedings.mlr.press/v70/guo17a.html.

JSON Schema Organization (2022). *JSON Schema: A Media Type for Describing JSON Documents*. URL: https://json-schema.org/draft/2020-12/json-schema-core.

National Institute of Standards and Technology (2024). *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile*. URL: https://doi.org/10.6028/NIST.AI.600-1.

OpenAI (2026a). *Decisions API*. URL: https://developers.openai.com/api/docs/guides/decisions.

— (2026b). *Error Codes*. URL: https://developers.openai.com/api/docs/guides/error-codes.

— (2026c). *Rate Limits*. URL: https://developers.openai.com/api/docs/guides/rate-limits.

— (2026d). *Responses API*. URL: https://developers.openai.com/api/docs/guides/text?lang=python.

— (2026e). *Structured Outputs*. URL: https://developers.openai.com/api/docs/guides/structured-outputs.

Pydantic (2026). *Models and JSON Schema*. URL: https://docs.pydantic.dev/latest/concepts/models/.